

# JAVA FX

---

UNIT-III ADVANCE JAVA

# TOPICS TO BE DISCUSSED

---

- Introduction to JavaFX? JavaFX Architecture
- JavaFX Scene, Graph
- Effects: Drop Shadow, Inner Shadow
- Color Adjust, Lighting and Reflection
- JavaFX Controls and layouts.
- JavaFX Program Structure Using FXML



# JAVAFX -INTRODUCTION

---

- JavaFX is a Java library utilized for creating Desktop applications and Rich Internet Applications (RIA).
- JavaFX applications can function across various platforms, such as Web, Mobile, and Desktops.
- JavaFX library includes Fundamentals, 2D Shapes, 3D Shapes, Effects, Animation, Text, Layouts, UI Controls, Transformations, Charts, JavaFX with CSS, and JavaFX with Media.

# JAVA FX INTRODUCTION

---

- JavaFX is designed to take over from swing in Java applications as a graphical user interface framework.
- Nevertheless, it offers a wider range of features compared to swing. Similar to Swing, JavaFX also offers its own elements and is not reliant on the operating system.
- It is light in weight and accelerated by hardware.
- It is compatible with different operating systems like Windows, Linux, and Mac OS.

# HISTORY

- 
- **Chris Oliver** was the developer of JavaFX.
  - The project was originally called **Form Follows Functions (F3)** at the beginning.
  - The aim is to offer more advanced features for developing GUI applications.
  - In June 2005, Sun Micro-systems took over the F3 project and rebranded it as JavaFX.
  - Sun Microsystems made the announcement at the W3 Conference in 2007.
  - **JavaFX 1.0** was launched in October 2008.
  - In 2009, ORACLE corporation purchased Sun Micro-Systems and launched JavaFX 1.2.
  - The most recent release of JavaFX is JavaFX 1.8, which came out on March 18, 2014.

# FEATURES OF JAVAFX

---

Feature	Description
Java Library	It is a Java library which consists of many classes and interfaces that are written in Java.
FXML	FXML is the XML based Declarative mark up language. The coding can be done in FXML to provide the more enhanced GUI to the user.
Scene Builder	Scene Builder generates FXML mark-up which can be ported to an IDE.
Web view	Web pages can be embedded with JavaFX applications. Web View uses WebKitHTML technology to embed web pages.
Built in UI controls	JavaFX contains Built-in components which are not dependent on operating system. The UI component are just enough to develop a full featured application.
CSS like styling	JavaFX code can be embedded with the CSS to improve the style of the application. We can enhance the view of our application with the simple knowledge of CSS.

# FEATURES OF JAVAFX

Feature	Description
Swing interoperability	The JavaFX applications can be embedded with swing code using the Swing Node class. We can update the existing swing application with the powerful features of JavaFX.
Canvas API	Canvas API provides the methods for drawing directly in an area of a JavaFX scene.
Rich Set of APIs	JavaFX provides a rich set of API's to develop GUI applications.
Integrated Graphics Library	An integrated set of classes are provided to deal with 2D and 3D graphics.
Graphics Pipeline	JavaFX graphics are based on Graphics rendered pipeline(prism). It offers smooth graphics which are hardware accelerated.
High Performance Media Engine	The media pipeline supports the playback of web multimedia on a low latency. It is based on a Gstreamer Multimedia framework.
Self-contained application deployment model	Self Contained application packages have all of the application resources and a private copy of Java and JavaFX Runtime.

# JAVAFX WITH NETBEANS

---

- Apache NetBeans is a free and open source integrated development environment (IDE) to develop applications using Java and it allows you to create applications using JavaFX.
- **NetBeans18** provides inbuilt support for JavaFX. On installing this, you can create a JavaFX application without any additional plugins or JAR files.
- Follow the steps that are given below to set up NetBeans for JavaFX environment
- **Step1:** Download and Install Apache NetBeans IDE 18 or later version.
- **Step2 :** Once you launch the NetBeans IDE, you will see the start page -> In the file menu, select **New Project...** to open the New project wizard.



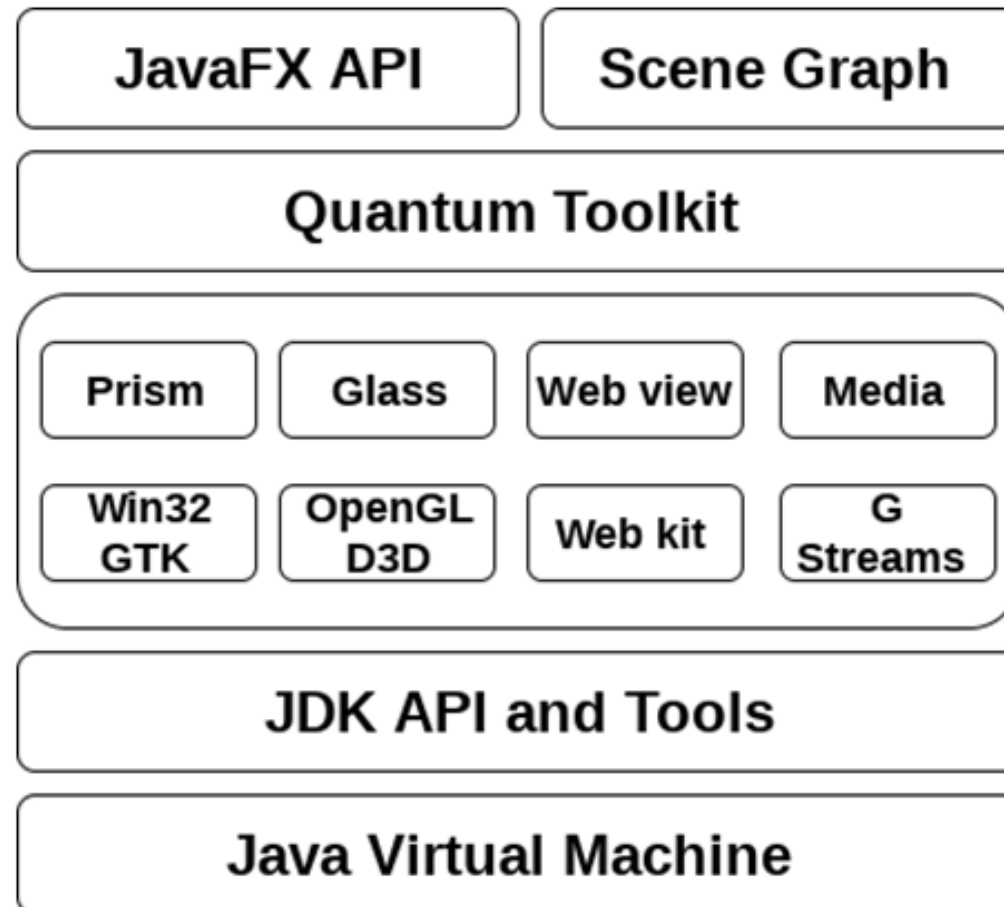
# JAVAFX WITH NETBEANS

---

- **Step 3:**In the **New Project** wizard, select **Java with Ant** and click on **Next**. It starts creating a new Java Application for you.
- **Step 4:** Select the name of the project and location of the project in the **New Java Application** window and then click **Finish**. It creates a sample application with the given name.
- In this instance, an application with a name **SampleJavaApplication** is created. Within this application, the NetBeans IDE will generate a Java program with the name **SampleJavaApplication.java**
- **Step 5:** Write JavaFX code ->Right-click on the file and select **Run Project** to run this code

# JAVAFX ARCHITECTURE

---



# JAVAFX ARCHITECTURE

---

- **JavaFX public API** -The upper level of JavaFX structure includes a JavaFX public API that includes essential classes for running a complete JavaFX application.
- **Scene Graph**- Building a JavaFX application begins here. It's a tree structure made up of nodes that depict every visual component of the user interface. It also possesses the ability to manage events. Overall, a scene graph is a gathering of nodes. Every node possesses its own unique identifier, appearance, and size. Each node in a scene graph is limited to having just one parent and can have multiple children.

# JAVAFX ARCHITECTURE

---

- **Graphics Engine** -The graphics support for the scene graph is given by the JavaFX graphics engine. It essentially provides support for both 2D and 3D graphics. It offers software rendering when the system's graphics hardware cannot support hardware accelerated rendering.
- The JavaFX has two pipelines for accelerated graphics.
- 1. **Prism** -A prism can be viewed as a high-performance graphics pipeline aided by hardware acceleration. It is able to display images in both 2D and 3D formats. Prism offers various methods for displaying graphics across various platforms.
  - DirectX 9 on windows XP or vista
  - DirectX 11 on windows 7
  - OpenGL on Mac, Linux and embedded
  - Java 2D when hardware acceleration is not possible

# JAVAFX ARCHITECTURE

---

- 2. Glass Windowing tool kit**-It can be found at the bottom of the JavaFX graphics stack. It essentially functions as an interface between the JavaFX platform and the native operating system, acting as a platform dependent layer. It is in charge of providing the operating system with services like handling windows, timers, event queues, and surfaces.
- **Quantum Tool kit**- The Quantum Tool Kit combines prism and glass windowing tool kit to make them accessible for the layers above in the stack.
  - **Web View** - We have the option to incorporate the HTML content into a JavaFX scene graph as well. JavaFX utilizes a component known as web view for this particular task. Web view utilizes web kit, an internal open source browser that can display HTML5, DOM, CSS, SVG, and JavaScript. By utilizing web view, we have the capability to display HTML content in a JavaFX application and customize the user interface with CSS styles.

# JAVAFX ARCHITECTURE

---

- **Media Engine-**By using Media engine, the JavaFX application can support the playback of audio and video media files. JavaFX media engine depends upon an open source engine called as G Streamer. The package **javafx.scene.media** contains all the classes and interfaces that can provide media functionalities to JavaFX applications.
-

# JAVAFX APPLICATION STRUCTURE

---

- JavaFX application is structured in a hierarchical way, consisting of three primary parts: Stage, Scene, and nodes. Importing `javafx.application.Application` class is necessary in all JavaFX applications. This offers the following life cycle techniques for JavaFX program.
- **`public void init()`**
- **`public abstract void start(Stage primaryStage)`**
- **`public void stop()`**

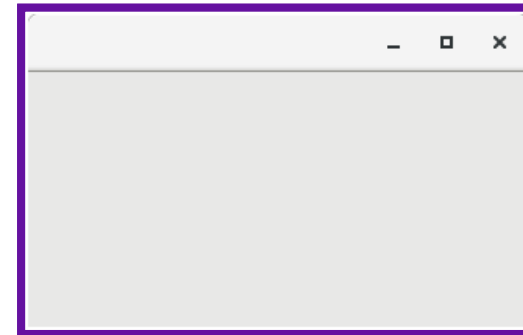
In order to create a basic JavaFX application, we need to:

1. Import **`javafx.application.Application`** into our code.
2. Inherit **`Application`** into our class.
3. Override **`start()`** method of Application class.

# STAGE

---

- In a JavaFX application, a Stage is analogous to a Frame in a Swing Application. It serves as a holding space for all the JavaFX items.
- The platform internally generates the Primary Stage. Additional phases can also be generated through the use of the software.
- The primary stage object is given to the start method. In order to display our primary stage, we must invoke the show method on the primary stage object.
- At first, the main Stage appears as shown.





# STAGE

---

- we are able to include different items on this initial platform.
- The items must be added hierarchically, meaning the scene graph must first be added to the primary Stage before any nodes can be added to the scene graph.
- A node can be any element in the user interface, such as text areas, buttons, shapes, and media.

# SCENE

---

- Scene is where all the actual physical elements (nodes) of a JavaFX application are contained. The `javafx.scene.Scene` class offers all the necessary methods for managing a scene object.
- Developing a setting is essential to help imagine the elements within the performance area. At any given moment, the scene object can solely be included in one stage.
- To integrate Scene in our JavaFX program, we need to bring in the `javafx.scene` package into our code.
- Creating the Scene involves instantiating the Scene class and providing the layout object to its constructor. Later in detail, we will delve into the Scene class and its method.

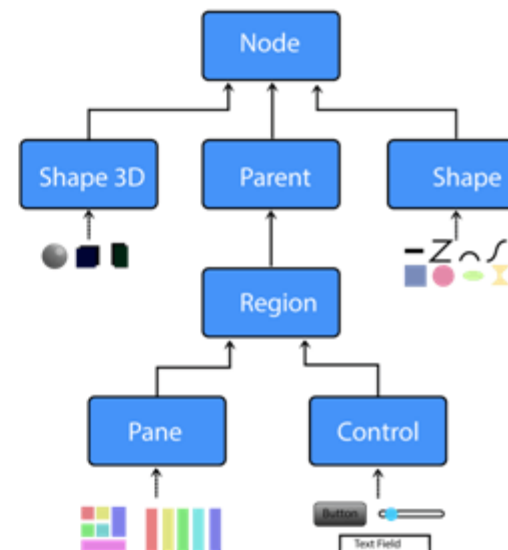
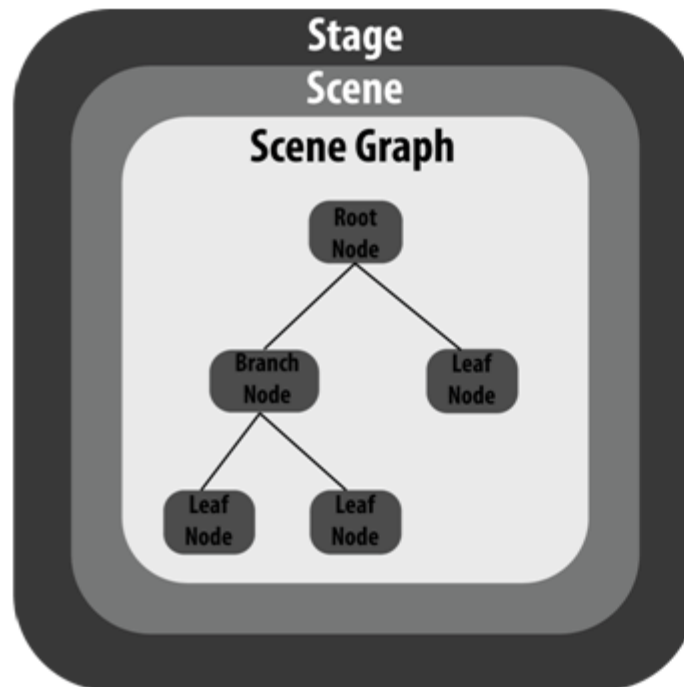
# SCENE GRAPH

---

- Scene Graph is present at the bottommost level of the hierarchy. It can be viewed as a gathering of different nodes.
- A node is the object that is displayed on the stage. It could be any button, text box, layout, image, radio button, check box, etc.
- A tree-like structure is used to implement the nodes.
- The scene graph always has one root.
- This will serve as the root node for all the other nodes in the scene graph.
- Yet, this particular node has the flexibility to be any of the available layouts within the JavaFX system.

# SCENE GRAPH

- The bottom level of the tree hierarchy contains the leaf nodes. E
- every node in scene graphs corresponds to classes in the javafx.scene package, so importing the package is necessary to develop a complete javafx application.



# FIRST JAVAFX APPLICATION

---

- **Step 1: Extend `javafx.application.Application` and override `start()`**
- **`start()`** method is the starting point of constructing a JavaFX application therefore we need to first override start method of **`javafx.application.Application`** class.
- Object of the class **`javafx.stage.Stage`** is passed into the **`start()`** method therefore import this class and pass its object into start method.
- **`JavaFX.application.Application`** needs to be imported in order to override start method.

## STEP 1: EXTEND JAVAFX.APPLICATION.APPLICATION AND OVERRIDE START()

---

```
package application;

import javafx.application.Application;
import javafx.stage.Stage;

public class Hello_World extends Application{

    @Override

    public void start(Stage primaryStage) throws Exception {

        // TODO Auto-generated method stub

    }

}
```

## STEP 2: CREATE A BUTTON

---

- **Step 2: Create a Button**
- A button can be created by instantiating the **`javafx.scene.control.Button`** class.
- For this, we have to import this class into our code.
- Pass the button label text in Button class constructor. The code will look like following.

## STEP 2: CREATE A BUTTON

---

```
package application;
import javafx.application.Application;
import javafx.scene.control.Button;
import javafx.stage.Stage;
public class Hello_World extends Application{
    @Override
    public void start(Stage primaryStage) throws Exception {
        // TODO Auto-generated method stub
        Button btn1=new Button("Say, Hello World");
    }
}
```



## STEP 3: CREATE A LAYOUT AND ADD BUTTON TO IT

---

- **Step 3: Create a layout and add button to it**
- JavaFX provides the number of layouts. We need to implement one of them in order to visualize the widgets properly.
- It exists at the top level of the scene graph and can be seen as a root node. All the other nodes (buttons, texts, etc.) need to be added to this layout.
- In this application, we have implemented **StackPane** layout. It can be implemented by instantiating **`javafx.scene.layout.StackPane`** class.
- The code will now look like following.

## STEP 3: CREATE A LAYOUT AND ADD BUTTON TO IT

package application;

---

```
import javafx.application.Application;
import javafx.scene.control.Button;
import javafx.stage.Stage;
import javafx.scene.layout.StackPane;
public class Hello_World extends Application{
    @Override
    public void start(Stage primaryStage) throws Exception {
        // TODO Auto-generated method stub
        Button btn1=new Button("Say, Hello World");
        StackPane root=new StackPane();
        root.getChildren().add(btn1);
    } }
```

## STEP 4: CREATE A SCENE

---

- The layout needs to be added to a scene. Scene remains at the higher level in the hierarchy of application structure.
- It can be created by instantiating **javafx.scene.Scene** class. We need to pass the layout object to the scene class constructor.
- Our application code will now look like following.

```
package application;  
  
import javafx.application.Application;  
  
import javafx.scene.Scene;  
  
import javafx.scene.control.Button;  
  
import javafx.stage.Stage;  
  
import javafx.scene.layout.StackPane;
```

## STEP 4: CREATE A SCENE

---

```
public class Hello_World extends Application{  
    @Override  
    public void start(Stage primaryStage) throws Exception {  
        // TODO Auto-generated method stub  
        Button btn1=new Button("Say, Hello World");  
        StackPane root=new StackPane();  
        root.getChildren().add(btn1);  
        Scene scene=new Scene(root);  
    } }
```

## STEP 5: PREPARE THE STAGE

---

- **javafx.stage.Stage** class provides some important methods which are required to be called to set some attributes for the stage.
- We can set the title of the stage.
- We also need to call `show()` method without which, the stage won't be shown.
- Lets look at the code which describes how can be prepare the stage for the application.

## STEP 5: PREPARE THE STAGE

---

```
package application;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.stage.Stage;
import javafx.scene.layout.StackPane;

public class Hello_World extends Application{

    @Override

    public void start(Stage primaryStage) throws Exception {
```

## STEP 5: PREPARE THE STAGE

---

```
Button btn1=new Button("Say, Hello World");  
    StackPane root=new StackPane();  
    root.getChildren().add(btn1);  
    Scene scene=new Scene(root);  
    primaryStage.setScene(scene);  
    primaryStage.setTitle("First JavaFX Application");  
    primaryStage.show();  
}  
  
}
```

## STEP 6: CREATE AN EVENT FOR THE BUTTON

---

- As our application prints hello world for an event on the button.
- We need to create an event for the button.
- For this purpose, call **setOnAction()** on the button and define a anonymous class Event Handler as a parameter to the method.
- Inside this anonymous class, define a method handle() which contains the code for how the event is handled. In our case, it is printing hello world on the console.

```
package application;
```

```
import javafx.application.Application;
```

```
import javafx.event.ActionEvent;
```

```
import javafx.event.EventHandler;
```

```
import javafx.scene.Scene;
```



## STEP 6: CREATE AN EVENT FOR THE BUTTON

---

```
import javafx.scene.control.Button;
import javafx.stage.Stage;
import javafx.scene.layout.StackPane;
public class Hello_World extends Application{
    @Override
    public void start(Stage primaryStage) throws Exception {
        // TODO Auto-generated method stub
        Button btn1=new Button("Say, Hello World");
        btn1.setOnAction(new EventHandler<ActionEvent>() {
```

## STEP 6: CREATE AN EVENT FOR THE BUTTON

---

@Override

```
    public void handle(ActionEvent arg0) {  
        System.out.println("hello world");  
    }  
}  
  
StackPane root = new StackPane();  
root.getChildren().add(btn1);  
Scene scene = new Scene(root, 600, 400);  
primaryStage.setScene(scene);  
primaryStage.setTitle("First JavaFX Application");  
primaryStage.show();  
}
```

## STEP 7: CREATE THE MAIN METHOD

---

- Till now, we have configured all the necessary things which are required to develop a basic JavaFX application but this application is still incomplete.
- We have not created main method yet.
- Hence, at the last, we need to create a main method in which we will launch the application i.e. will call `launch()` method and pass the command line arguments (`args`) to it.
- The code will now look like following.

## STEP 7: CREATE THE MAIN METHOD

---

```
package application;  
  
import javafx.application.Application;  
  
import javafx.event.ActionEvent;  
  
import javafx.event.EventHandler;  
  
import javafx.scene.Scene;  
  
import javafx.scene.control.Button;  
  
import javafx.stage.Stage;  
  
import javafx.scene.layout.StackPane;  
  
public class Hello_World extends Application{
```

## STEP 7: CREATE THE MAIN METHOD

---

@Override

```
public void start(Stage primaryStage) throws Exception {  
    Button btn1=new Button("Say, Hello World");  
    btn1.setOnAction(new EventHandler<ActionEvent>() {
```

@Override

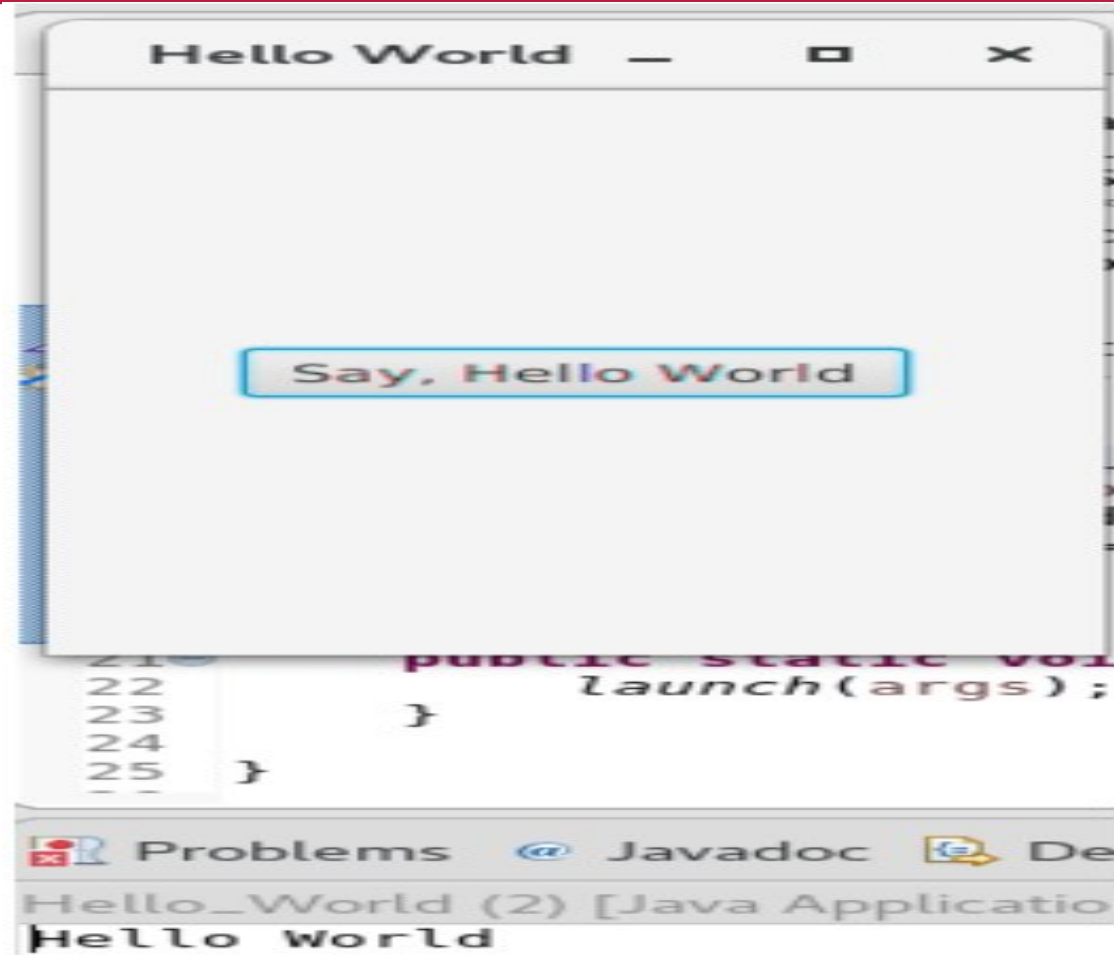
```
        public void handle(ActionEvent arg0) {  
            // TODO Auto-generated method stub  
            System.out.println("hello world");  
        }  
    });
```

## STEP 7: CREATE THE MAIN METHOD

---

```
StackPane root=new StackPane();
    root.getChildren().add(btn1);
    Scene scene=new Scene(root,600,400);
    primaryStage.setTitle("First JavaFX Application");
    primaryStage.setScene(scene);
    primaryStage.show();
}
publicstaticvoid main (String[] args)
{
    launch(args);
} }
```

# OUTPUT



# JAVAFX 2D SHAPES

---

- JavaFX provides the flexibility to create our own 2D shapes on the screen .
- There are various classes which can be used to implement 2D shapes in our application. All these classes resides in **javafx.scene.shape** package.
- This package contains the classes which represents different types of 2D shapes. There are several methods in the classes which deals with the coordinates regarding 2D shape creation.
- **In general, a two dimensional shape can be defined as the geometrical figure that can be drawn on the coordinate system consist of X and Y planes. However, this is different from 3D shapes in the sense that each point of the 2D shape always consists of two coordinates (X,Y).**



# HOW TO CREATE 2D SHAPES?

---

- Using JavaFX, we can create 2D shapes such as Line, Rectangle, Circle, Ellipse, Polygon, Cubic Curve, quad curve, Arc, etc.
- The class `javafx.scene.shape.Shape` is the base class for all the shape classes.
- For creating a two dimensional shape, the following instructions need to be followed:
  - 1. Instantiate the respective class: **Rectangle rect = new Rectangle()**
  - 2. Set the required properties for the class using instance setter methods.
  - 3. Add class object to the Group layout

# JAVAFX SHAPE CLASSES


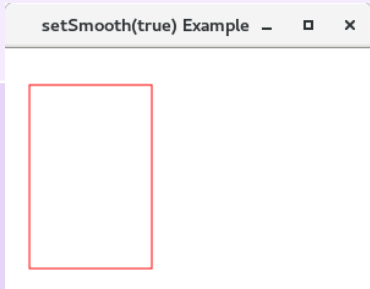
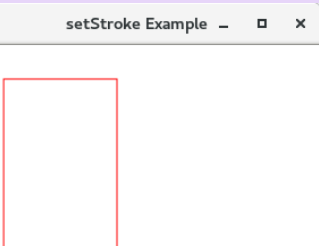
Shapes	Description
<u><a href="#">Line</a></u>	In general, Line is the geometrical figure which joins two (X,Y) points on 2D coordinate system. In JavaFX, <b><code>javafx.scene.shape.Line</code></b> class needs to be instantiated in order to create lines.
<u><a href="#">Rectangle</a></u>	In general, Rectangle is the geometrical figure with two pairs of two equal sides and four right angles at their joint. In JavaFX, <b><code>javafx.scene.shape.Rectangle</code></b> class needs to be instantiated in order to create Rectangles.
<u><a href="#">Ellipse</a></u>	In general, ellipse can be defined as a curve with two focal points. The sum of the distances to the focal points are constant from each point of the ellipse. In JavaFX, <b><code>javafx.scene.shape.Ellipse</code></b> class needs to be instantiated in order to create Ellipse.
<u><a href="#">Circle</a></u>	A circle is the special type of Ellipse having both the focal points at the same location. In JavaFX, Circle can be created by instantiating <b><code>javafx.scene.shape.Circle</code></b> class.

# JAVAFX SHAPE CLASSES

---

Shapes	Description
<u>Polygon</u>	Polygon is a geometrical figure that can be created by joining the multiple Co-planner line segments. In JavaFX, <b>javafx.scene.shape.Polygon</b> class needs to be instantiated in order to create polygon.
<u>Cubic Curve</u>	A Cubic curve is a curve of degree 3 in the XY plane. In Javafx, <b>javafx.scene.shape.CubicCurve</b> class needs to be instantiated in order to create Cubic Curves.
<u>Quad Curve</u>	A Quad Curve is a curve of degree 2 in the XY plane. In JavaFX, <b>javafx.scene.shape.QuadCurve</b> class needs to be instantiated in order to create QuadCurve.

# JAVAFX SHAPE PROPERTIES

Property	Description	Method	Example
fill	Used to fill the shape with a defined paint. This is a object <paint> type property.	setFill(Paint)	
smooth	This is a boolean type property. If true is passes then the edges of the shape will become smooth.	setSmooth(boolean)	
strokeDashOffset	It defines the distances in the coordinate system which shows the shapes in the dashing patterns. This is a double type property.	setStrokeDashOffset(Double)	
strokeLineCap	It represents the style of the line end cap. It is a strokeLineCap type property.	setStrokeLineCap(StrokeLineCap)	

# JAVAFX SHAPE PROPERTIES

Property	Description	Method
strokeLineJoin	It represents the style of the joint of the two paths.	setStrokeLineJoin(StrokeLineJoin)
strokeMiterLimit	It applies the limitation on the distance between the inside and outside points of a joint. It is a double type property.	setStrokeMiterLimit(Double)
stroke	It is a colour type property which represents the colour of the boundary line of the shape.	setStroke(Paint)
strokeType	It represents the type of the stroke (where the boundary line will be imposed to the shape) whether inside, outside or centred.	setStrokeType(StrokeType)
strokeWidth	It represents the width of the stroke.	setStrokeWidth(Double)

# EXAMPLE- LINE

---

```
package application;

import javafx.application.Application;

import javafx.scene.Group;

import javafx.scene.Scene;

import javafx.scene.paint.Color;

import javafx.scene.shape.Line;

import javafx.stage.Stage;

public class LineDrawingExamples extends Application{

    public static void main(String[] args) {

        launch(args);

    }

}
```

@Override

public void start(Stage primaryStage) throws Exception {

---

primaryStage.setTitle("Line Drawing Examples");

Line line1 = new Line(10,50,150,50); //Line(startX,startY,endX,endY)

Line line2 = new Line(10,100,150,100);

Line line3 = new Line(10,50,10,100);

Line line4 = new Line(150,50,150,100);

Group root = new Group();

root.getChildren().addAll(line1,line2,line3,line4);

Scene scene = new Scene (root,300,200,Color.GREEN);

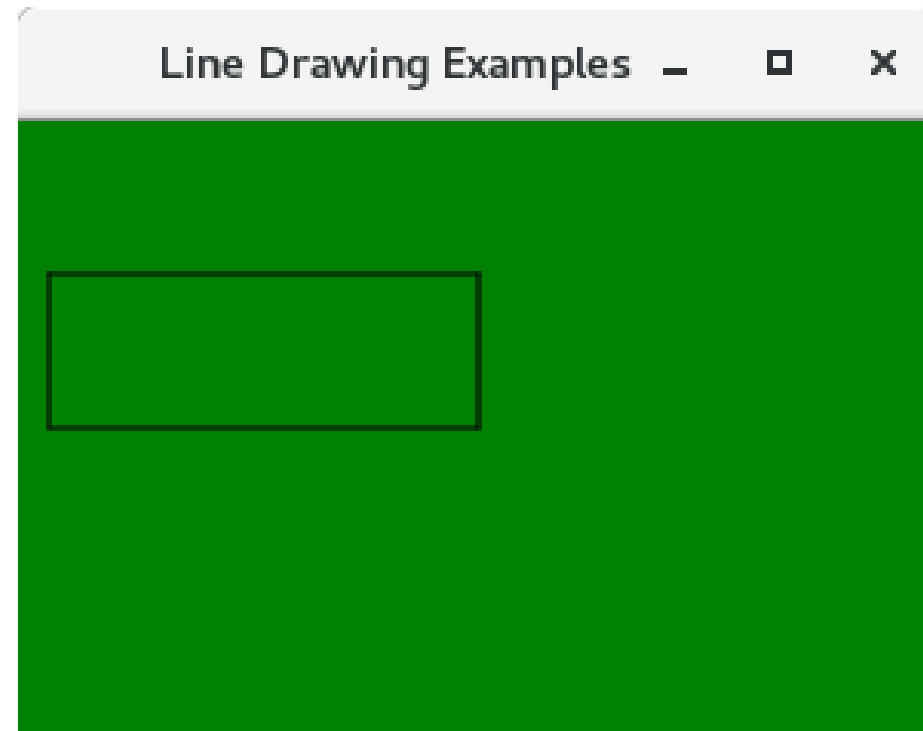
primaryStage.setScene(scene);

primaryStage.show();

} }

# OUTPUT

---





# EXAMPLE- ROUNDED RECTANGLE

```
public class Shape_Example extends Application{  
    @Override  
    public void start(Stage primaryStage) throws Exception {  
        // TODO Auto-generated method stub  
        primaryStage.setTitle("Rectangle Example");  
        Group group = new Group();  
        Rectangle rect=new Rectangle();  
        rect.setX(20);  
        rect.setY(20);  
        rect.setWidth(100);  
        rect.setHeight(100);  
    }  
}
```

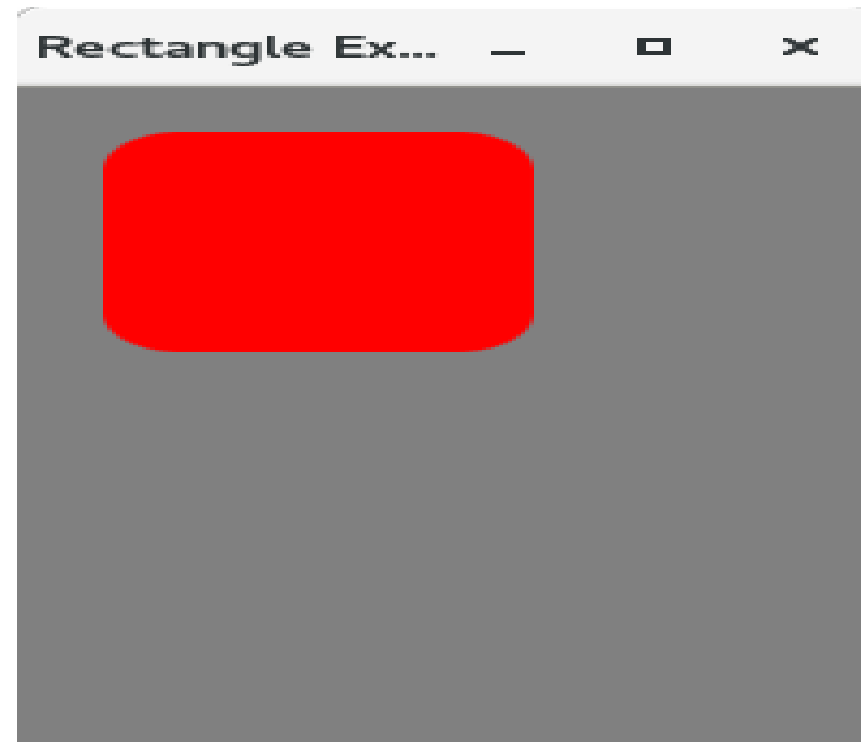
```
rect.setArcHeight(35);  
rect.setArcWidth(35);  
rect.setFill(Color.RED);
```

---

```
group.getChildren().addAll(rect);  
    Scene scene = new Scene(group,200,300,Color.GRAY);  
    primaryStage.setScene(scene);  
    primaryStage.show();  
}  
public static void main(String[] args) {  
    launch(args);  
}  
}
```

# OUTPUT

---



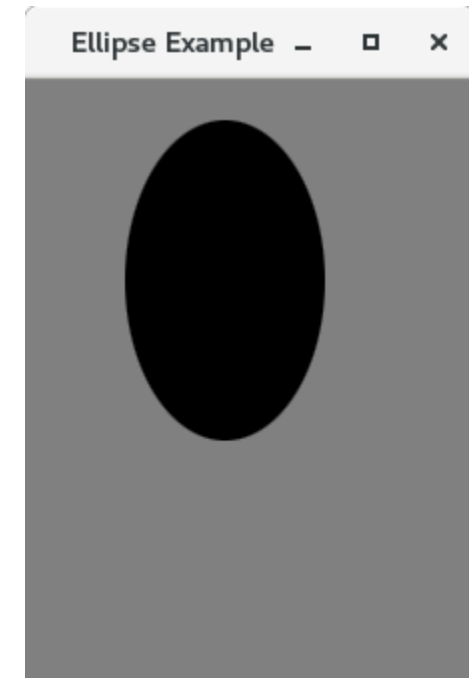
# ELLIPSE

---

```
public class Shape_Example extends Application{  
    @Override  
    public void start(Stage primaryStage) throws Exception {  
        // TODO Auto-generated method stub  
        primaryStage.setTitle("Ellipse Example");  
        Group group = new Group();  
        Ellipse ellipse = new Ellipse();  
        ellipse.setCenterX(100);  
        ellipse.setCenterY(100);  
        ellipse.setRadiusX(50);  
        ellipse.setRadiusY(80);  
    }  
}
```

# ELLIPSE

```
group.getChildren().addAll(ellipse);  
    Scene scene = new Scene(group,200,300,Color.GRAY);  
    primaryStage.setScene(scene);  
    primaryStage.show();  
}  
public static void main(String[] args) {  
    launch(args);  
}  
  
}
```



# JAVAFX TEXT

---

- In some of the cases, we need to provide the text based information on the interface of our application.
- JavaFX library provides a class named **javafx.scene.text.Text** for this purpose. This class provides various methods to alter various properties of the text.
- **Creating a text node**
- Use the setter method **setText(string)** to set the string as a text for the text class object. Syntax is as follows:

```
Text <text_Object> = new Text();
```

```
text.setText(<string-text>);
```

# PROPERTIES OF TEXT

Property	Description	Setter Methods
font	Font of the text.	setFont(Font value)
linespacing	Vertical space in pixels between the lines. It is double type property.	setLineSpacing(double spacing)
strikethrough	This is a boolean type property. We can put a line through the text by setting this property to true.	setStrikeThrough(boolean value)
textalignment	Horizontal Text alignment	setTextAlignment(TextAlignment value)
textorigin	Origin of text coordinate system in local coordinate system.	setTextOrigin(VPos value)
text	It is a string type property. It defines the text string which is to be displayed.	setText(String value)
underline	It is a boolean type property. We can underline the text by setting this property to true.	setUnderLine(boolean value)

# PROPERTIES OF TEXT

Property	Description	Setter Methods
wrapwidth	Width limit for the text from where the text is to be wrapped. It is a double type property.	setWidth(double value)
x	X coordinate of the text	setX(double value)
y	Y coordinate of the text	setY(double value)
boundtype	This property is of object type. It determines the way in which the bounds of the text is being calculated.	setBoundsType(TextBoundsType value)
fontsmoothingType	Defines the requested smoothing type for the font.	setFontSmoothingType(FontSmoothingType value)



# FONT POSITION OF THE TEXT

---

- JavaFX enables us to apply various fonts to the text nodes. We can set the property **font** of the Text class by using the setter method **setFont()**. This method accepts the object of **Font** class.
- The method **Font.font()** accepts the following parameters.
  - 1.Family:** it represents the family of the font. It is of string type and should be an appropriate font family present in the system.
  - 2.Weight:** this Font class property is for the weight of the font. There are 9 values which can be used as the font weight. The values are **FontWeight.BLACK, BOLD, EXTRA\_BOLD, EXTRA\_LIGHT, LIGHT, MEDIUM, NORMAL, SEMI\_BOLD, THIN.**

# FONT POSITION OF THE TEXT

---

3. **Posture:** this Font class property represents the posture of the font. It can be either **FontPosture.ITALIC** or **FontPosture.REGULAR**.
4. **Size:** this is a double type property. It is used to set the size of the font.

- **Syntax**

```
<text_object>.setFont(Font.font(<String font_family>, <FontWeight>, <FontPosture>,  
<FontSize>)
```

# EXAMPLE-TEXT DISPLAY

---

```
public void start(Stage primaryStage) throws Exception {  
    Text text = new Text();  
    text.setX(100);  
    text.setY(20);  
    text.setFont(Font.font("Cambria", FontWeight.BOLD, FontPosture.REGULAR, 25));  
    text.setFill(Color.BLUE); // setting colour of the text to blue  
    text.setStroke(Color.BLACK); // setting the stroke for the text  
    text.setStrokeWidth(1); // setting stroke width to 2  
    text.setText("Welcome to JavaFX");  
}
```

```
Group root = new Group();  
Scene scene = new Scene(root,500,200);  


---

  
root.getChildren().add(text);  
primaryStage.setScene(scene);  
primaryStage.setTitle("Text Example");  
primaryStage.show();  
}  
public static void main(String[] args) {  
    launch(args);  
}  
}
```

# JAVAFX EFFECTS

---

- Effects are basically actions that can improve the appearance of the graphics.
- JavaFX provides the package named as **javafx.scene.effect** which contains various classes that can be used to apply effects on the UI graphic components like images and shapes.
- JavaFX provides a method named as **setEffect()** which needs to be called through a node object. We need to pass the effect class object into this method. To apply any effect to the node, follow the following steps.

1. Create the node
2. Create the object of the respective Effect class which is to be applied on the node.
3. Set the properties of the Effect.
4. Call **setEffect()** method through the node object and pass the Effect class object into it.

# JAVAFX EFFECTS

Effects	Description
<u><a href="#">ColorAdjust</a></u>	This Effect adjusts the color of the node by varying the properties like Hue, Saturation, Brightness, contrast etc. <b>javafx.scene.effect.ColorAdjust</b> class deals with all the stuff regarding adjustments of colors of the node.
<u><a href="#">ColorInput</a></u>	<b>javafx.scene.ColorInput</b> class represents ColorInput effect. It makes a coloured rectangle. This displays a rectangular box if applied to a node
<u><a href="#">ImageInput</a></u>	<b>ImageInput</b> effect is used to bind the image to the scene. It basically passes the specified image to some effect.
<u><a href="#">Reflection</a></u>	It adds the reflection of the node on the bottom of the node. The class named as <b>javafx.scene.effect.Reflection</b> represents Reflection effect.
<u><a href="#">Shadow</a></u>	This duplicates the nodes with the blurry edges. The class named as <b>javafx.scene.effect.Shadow</b> represents Shadow effect.

# JAVAFX EFFECTS

---

Effects	Description
<u><a href="#">DropShadow</a></u>	This is a high level effect that is used to display the duplicate content behind the original content with the specified color and size.
<u><a href="#">InnerShadow</a></u>	This effect displays the shadow inside the edges of the nodes to which it is applied.
<u><a href="#">Lighting</a></u>	This effect is used to lighten the node from a light source. This effect is represented by <b><code>javafx.scene.effect.Lighting</code></b> class.
<u><a href="#">Glow</a></u>	This effect is very much similar to Bloom. This can make the input image glow by enhancing the brightness of the bright pixels.

# JAVAFX REFLECTION EFFECT

---

- Reflection can be defined as the change in the direction.
  - JavaFX allows us to generate the reflection effect on any node.
  - Reflection effect basically adds the reflection of the node to its bottom.
  - It is represented by the class **`javafx.scene.effect.Reflection`**.
  - The class contains two constructors.
- 1.**`public Reflection()`** : Creates a new instance of Reflection with the default parameters
  - 2.**`public Reflection(double topOffset, double fraction, double topOpacity, double bottomOpacity)`** : Creates a new instance of Reflection with the specified parameters



# JAVAFX REFLECTION EFFECT- EXAMPLE

---

```
public void start(Stage primaryStage) throws Exception {  
    Text text = new Text();  
    text.setFont(Font.font("calibri",FontWeight.BLACK,FontPosture.REGULAR,20));  
    text.setText("Welcome to JavaFX");  
    text.setX(90);  
    text.setY(90);  
    Reflection ref = new Reflection();  
    ref.setBottomOpacity(0.2);  
    ref.setFraction(12);  
}
```

# JAVAFX REFLECTION EFFECT- EXAMPLE

---

```
ref.setTopOffset(10);  
    ref.setTopOpacity(0.2);  
    text.setEffect(ref);  
    Group root = new Group();  
    Scene scene = new Scene(root,400,300);  
    root.getChildren().add(text);  
    primaryStage.setScene(scene);  
    primaryStage.setTitle("Reflection Example");  
    primaryStage.show();  
}
```

# JAVAFX DROPSHADOW EFFECT

---

- This effect is similar to the shadow effect and the duplicate of the node is displayed behind the original node with the specified size and color.
- The class **`javafx.scene.effect.DropShadow`** represents the Drop Shadow effect.
- The class contains four constructors
  - 1.**`public DropShadow()`** : It creates the instance with the default parameters.
  - 2.**`public DropShadow(double radius, Color color)`** : It creates the instance with the specified radius and color values.
  - 3.**`public DropShadow(double radius, double offsetX, double offsetY, Color color)`** : It creates the instance with the specified radius, offset and color values.
  - 4.**`public DropShadow(BlurType blurtype, Color color, double radius, double spread, double offsetX, double offsetY)`** : It creates the instance with the specified BlurType, color, radius, spread and offset values.

# JAVAFX DROPShadow EFFECT- EXAMPLE

---

```
public void start(Stage primaryStage) throws Exception {
```

```
    Image img = new Image("logo.png");
```

```
    ImageView imgview = new ImageView(img);
```

```
    imgview.setX(130);
```

```
    imgview.setY(125);
```

```
    imgview.setFitWidth(175);
```

```
    DropShadow drop = new DropShadow();
```

```
    drop.setBlurType(BlurType.GAUSSIAN);
```

# JAVAFX DROPShadow EFFECT- EXAMPLE

---

```
drop.setColor(Color.BLUE);  
drop.setHeight(100);  
drop.setWidth(150);  
drop.setOffsetX(10);  
drop.setOffsetY(10);  
drop.setSpread(0.2);  
drop.setRadius(10);  
imgview.setEffect(drop);  
Group root = new Group();
```

# JAVAFX DROPShadow EFFECT- EXAMPLE

---

```
Scene scene = new Scene(root,400,300);  
    root.getChildren().add(imgview);  
    primaryStage.setScene(scene);  
    primaryStage.setTitle("DropShadow Example");  
    primaryStage.show();  
}  
public static void main(String[] args) {  
    launch(args);  
}  
}
```

# JAVAFX INNERSHADOW EFFECT

---

- In this effect, the shadow is displayed inside the edges of the node.
  - The class `javafx.scene.effect.InnerShadow` represents the InnerShadow effect.
1. `public InnerShadow()` : creates the instance with the default parameters.
  2. `public InnerShadow(double radius, Color color)` : creates the instance with the specified radius and color value.
  3. `public InnerShadow(double radius, double offsetX, double offsetY, Color color)` : creates the instance with the specified radius, offset and color values.
  4. `public InnerShadow(BlurType blurtype, Color color, double radius, double choke, double offsetX, double offsetY)` : creates the instance with the specified BlurType. Color, Radius, Choke and offset values.

# JAVAFX INNERSHADOW EFFECT- EXAMPLE

---

```
public class ShadowExample extends Application{  
    @Override  
    public void start(Stage primaryStage) throws Exception {  
        Image img = new Image("logo.png");  
        ImageView imgview = new ImageView(img);  
        imgview.setFitHeight(100);  
        imgview.setFitWidth(350);  
        imgview.setX(100);  
        imgview.setY(100);  
    }  
}
```



# JAVAFX INNERSHADOW EFFECT- EXAMPLE

---

```
InnerShadow shadow = new InnerShadow();  
shadow.setBlurType(BlurType.GAUSSIAN);  
shadow.setColor(Color.RED);  
shadow.setHeight(25);  
shadow.setRadius(12);  
shadow.setWidth(20);  
shadow.setChoke(0.9);  
imgview.setEffect(shadow);  
Group root = new Group();  
root.getChildren().add(imgview);
```

# JAVAFX INNERSHADOW EFFECT- EXAMPLE

---

```
Scene scene = new Scene(root,600,350);
    primaryStage.setScene(scene);
    primaryStage.setTitle("InnerShadow Effect Example");
    primaryStage.show();
}
public static void main(String[] args) {
    launch(args);
}
}
```

# JAVAFX LIGHTING EFFECT

---

- This effect is used to lighten a node from a light source.
  - There are various kinds of light sources i.e. Point, Distant and Spot.
  - The class **javafx.scene.effect.Lighting** represents the lighting effect.
  - The class contains two constructors.
- 1.**public Lighting()** : creates a new instance of Lighting with the default value of light source.
  - 2.**public Lighting(Light light)** : creates a new instance of Lighting with the specified value of light source.

# JAVAFX LIGHTING EFFECT EXAMPLE

---

```
public class LightingExample1 extends Application {  
    @Override  
    public void start(Stage stage) {  
        Text text = new Text();  
        text.setFont(Font.font(null, FontWeight.BOLD, 35));  
        text.setX(60);  
        text.setY(100);  
        text.setText("Welcome to JavaFX");  
        text.setFill(Color.GREEN);  
    }  
}
```

# JAVAFX LIGHTING EFFECT EXAMPLE

---

```
Image img = new Image("Rose.png");  
ImageView imgview = new ImageView(img);  
imgview.setX(150);  
imgview.setY(200);  
Lighting lighting = new Lighting();  
text.setEffect(lighting);  
imgview.setEffect(lighting);  
Group root = new Group(text,imgview);  
Scene scene = new Scene(root, 580, 420);
```

# JAVAFX LIGHTING EFFECT EXAMPLE

---

```
stage.setTitle("lighting effect example");  
stage.setScene(scene);  
stage.show();  
}  
public static void main(String args[]){  
    launch(args);  
}  
}
```

# JAVAFX COLORADJUST EFFECT

---

- JavaFX allows us to adjust the color of an image by adjusting the properties like hue, saturation, brightness and contrast of the color of image.
- The class **javafx.scene.effect.ColorAdjust** contains various properties and methods that can be used to apply the **ColorAdjust** effect on the node.
- The class contains two constructors given below.
  - 1.**public ColorAdjust()** : creates the new instance of ColorAdjust with the default parameters.
  - 2.**public ColorAdjust(double hue, double saturation, double brightness, double contrast)** : Creates the new instance of the ColorAdjust with the specified parameters.

# JAVAFX COLORADJUST EFFECT EXAMPLE

---

```
public class ColorAdjustEffect extends Application{  
    @Override  
    public void start(Stage primaryStage) throws Exception {  
        Image img1 = new Image("linux-first.png");  
        Image img2 = new Image("linux-first.png");  
        ImageView imgview1 = new ImageView(img1);  
        ImageView imgview2 = new ImageView(img2);  
        Text text1 = new Text();  
        Text text2 = new Text();
```



# JAVAFX COLORADJUST EFFECT EXAMPLE

---

```
text1.setText("ColorAdjust Effect Applied");  
text2.setText("ColorAdjust Effect Not Applied");  
text1.setX(50);  
text1.setY(300);  
text2.setX(355);  
text2.setY(300);  
text1.setFont(Font.font("Courier 10 Pitch",FontWeight.BOLD,FontPosture.REGULAR,16));  
text2.setFont(Font.font("Courier 10 Pitch",FontWeight.BOLD,FontPosture.REGULAR,16));
```

# JAVAFX COLORADJUST EFFECT EXAMPLE

---

```
text1.setFill(Color.RED);  
text2.setFill(Color.RED);  
text1.setStroke(Color.BLACK);  
    text2.setStroke(Color.BLACK);  
text1.setStrokeWidth(0.2);  
text2.setStrokeWidth(0.2);  
imgview1.setX(100);  
imgview1.setY(90);  
imgview2.setX(400);  
imgview2.setY(90);
```

# JAVAFX COLORADJUST EFFECT EXAMPLE

---

ColorAdjust c = **new** ColorAdjust(); // creating the instance of the ColorAdjust effect.

c.setBrightness(0.2); // setting the brightness of the color.

c.setContrast(0.1); // setting the contrast of the color

c.setHue(0.3); // setting the hue of the color

c.setSaturation(0.45); // setting the hue of the color.

imgview1.setEffect(c); //applying effect on the image

Group root = **new** Group();

root.getChildren().addAll(imgview1,imgview2,text1,text2);

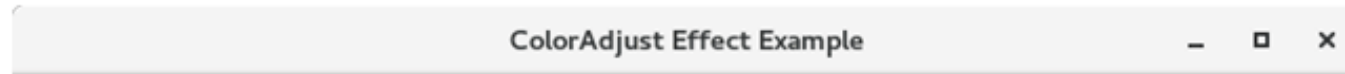
# JAVAFX COLORADJUST EFFECT EXAMPLE

---

```
Scene scene = new Scene(root,700,400);
    primaryStage.setScene(scene);
    primaryStage.setTitle("ColorAdjust Effect Example");
    primaryStage.show();
}
public static void main(String[] args) {
    launch(args);
} }
```

# JAVAFX COLORADJUST EFFECT EXAMPLE

---



**ColorAdjust Effect Applied**



**ColorAdjust Effect Not Applied**